Modifiers

To help you with what problems to try, problems marked with **[K]** are key questions that tests you on the core concepts, please do them first. Problems marked with **[H]** are harder problems that we recommend you to try after you complete all other questions (or perhaps you prefer a challenge). Good luck!!!

Contents

§1 Recurrences

Exercise 1.1. [K] Determine the asymptotic growth rate of the solutions to the following recurrences. You may use the Master Theorem if it is applicable.

(a) $T(n) = 2T(n/2) + n(2 + \sin n)$.

(b) $T(n) = 2T(n/2) + \sqrt{n} + \log n$.

(c) $T(n) = 8T(n/2) + n^{\log_2 n}$.

(d) $T(n) = T(n-1) + n$.

Solution. **(a)** Here, we realise that $a = 2$, $b = 2$, and $f(n) = n(2 + \sin n)$. But $\sin n \leq 1$ for all n and so, $f(n) = \Theta(n)$. The critical exponent is

$$c^* = \log_b a = \log_2 2 = 1.$$

Thus, the second case of the Master Theorem applies and we get

$$T(n) = \Theta(n \log n).$$

(b) Again, we repeat the same process. We realise that $a = 2$, $b = 2$, and $f(n) = \sqrt{n} + \log n$. So, the critical exponent is $c^* = 1$. Since $\log n$ *eventually* grows slower than $\sqrt{n}$, we have that

$$f(n) = \Theta(\sqrt{n}). = \Theta\left(n^{1/2}\right).$$

This implies that

$$f(n) = O(n^{0.9}) = O(n^{c^*-0.1}),$$

so the first case of the Master Theorem applies and we obtain $T(n) = \Theta(n)$.

(c) Here, $a = 8$, $b = 2$, and $f(n) = n^{\log_2 n}$. So the critical exponent is

$$c^* = \log_b a = \log_2 8 = 3.$$

On the other hand, for large enough n, we have that $\log_2 n \geq 4$. So

$$f(n) = n^{\log_2 n} = \Omega(n^4).$$

Consequently,

$$f(n) = \Omega(n^{c^*+1}).$$

To be able to use the third case of the Master Theorem, we have to show that for some $0 < c < 1$, the following holds for sufficiently large n:

$$af\left(\frac{n}{b}\right) < cf(n).$$

In our case, this translates to

$$8\left(\frac{n}{2}\right)^{\log_2 \frac{n}{2}} < cn^{\log n}.$$

Now, we have

$$\begin{aligned} 8\left(\frac{n}{2}\right)^{\log_2 \frac{n}{2}} &= 8\left(\frac{n}{2}\right)^{\log_2 n - \log_2 2} \\ &= 8\left(\frac{n}{2}\right)^{\log_2 n - 1} \\ &< 8\left(\frac{n}{2}\right)^{\log_2 n} \\ &= \frac{8n^{\log_2 n}}{2^{\log_2 n}} \\ &= \frac{8}{n} n^{\log_2 n}. \end{aligned}$$

If $n > 16$, then

$$8\left(\frac{n}{2}\right)^{\log_2 \frac{n}{2}} < \frac{1}{2} n^{\log_2 n},$$

so the required inequality is satisfied with $c = \frac{1}{2}$ for all $n > 16$. Therefore, by case 3 of the Master Theorem, the solution is

$$T(n) = \Theta(f(n)) = \Theta\left(n^{\log_2 n}\right).$$

(d) Note that the Master Theorem does not apply; however, we can alter the proof of the Master Theorem to obtain the solution to the recurrence. For every k, we have $T(k) = T(k-1) + k$. So unrolling the recurrence, we obtain

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= \underbrace{[T(n-2) + (n-1)]}_{T(n-1)} + n \\ &= T(n-2) + (n-1) + n \\ &= [T(n-3) + (n-2)] + (n-1) + n \\ &= \dots \\ &= T(1) + (n-(n-2)) + (n-(n-3)) + \cdots + (n-1) + n \\ &= T(1) + (2 + 3 + \cdots + n) \\ &= T(1) + \frac{n(n+1)}{2} - 1 \\ &= \Theta(n^2). \end{aligned}$$

■

Exercise 1.2. [K] Explain why we cannot apply the Master Theorem to the following recurrences.

(a) $T(n) = 2^n T(n/2) + n^n$.

(b) $T(n) = T(n/2) - n^2 \log n$.

(c) $T(n) = \frac{1}{3} T(n/3) + n$.

(d) $T(n) = 3T(3n) + n$.

Solution. **(a)** The Master Theorem requires the value of a to be constant. Here, the value of $a = 2^n$ depends on the input size which is non-constant.

(b) The Master Theorem requires $f(n)$ to be non-negative. We can see that, for $n \in \mathbb{N}$, $f(n) \leq 0$. So, the Master Theorem cannot be applied.

(c) The Master Theorem requires $a \geq 1$. Here, $a = 1/3$ and so, the conditions of the Master Theorem are not met.

(d) The Master Theorem requires $b > 1$. However, we see that we can write $T(n)$ as

$$T(n) = 3T\left(\frac{n}{1/3}\right) + n;$$

in this case, we see that $b = 1/3 < 1$. So the conditions of the Master Theorem are not met.

■

Exercise 1.3. [H] Consider the following naive Fibonacci algorithm.

Algorithm 1.1 $F(n)$: The naive Fibonacci algorithm

```
Require: n ≥ 1
  if n = 1 or n = 2 then
      return 1
  else
      return F(n − 1) + F(n − 2)
  end if
```

When analysing its time complexity, this yields us with the recurrence $T(n) = T(n-1) + T(n-2)$. Show that this yields us with a running time of $\Theta(\varphi^n)$, where $\varphi = \frac{1+\sqrt{5}}{2}$ which is the golden ratio. How do you propose that we improve upon this running time?

> **Hint —** *This is a standard recurrence relation. Guess $T(n) = a^n$ and solve for a.*

Solution. We solve $T(n) = T(n-1) + T(n-2)$. Using the standard trick of solving recurrence relations, we guess $T(n) = a^n$ for some a. Then this produces the recurrence:

$$a^n = a^{n-1} + a^{n-2}.$$

Dividing both sides by a^{n-2}, we obtain the quadratic equation:

$$a^2 - a - 1 = 0.$$

Then the quadratic equation yields

$$a = \frac{1 \pm \sqrt{5}}{2}.$$

In other words, we obtain

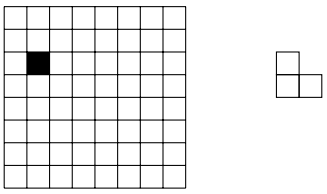
$$T(n) = A \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n + B \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n = \Theta\left(\varphi^n\right),$$

where $\varphi = \frac{1+\sqrt{5}}{2}$.

The inefficiency comes from the fact that we are making unnecessary and repeated calls to problems that we have already seen before. For example, to compute $F(n)$, we compute $F(n-1)$ and $F(n-2)$. To compute $F(n-1)$, we compute $F(n-2)$ again. Because these values never change, a way to improve the overall running time is to *cache* the results that we have computed previously and only make $O(1)$ calls for results that we have already computed. This reduces our complexity dramatically from exponential to $O(n)$. This is an example of *dynamic programming*, something we will see later down the track. ■

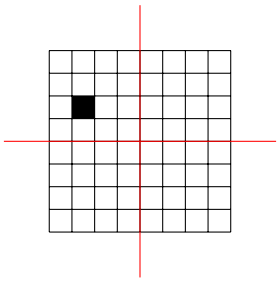
§2 Divide and Conquer

Exercise 2.1. [K] You are given a $2^n \times 2^n$ board with one of its cells missing (i.e., the board has a hole). The position of the missing cell can be arbitrary. You are also given a supply of "trominoes", each of which can cover three cells as below.

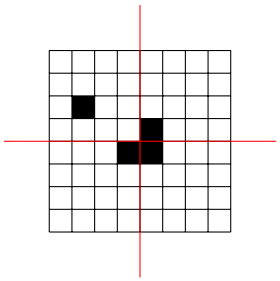


Design an algorithm that covers the entire board (except for the hole) with these "trominoes". Note that the trominoes can be rotated and your solution should not try to brute force all possible placement of those "trominoes".

Solution. We proceed by a divide and conquer recursion; thus, we split the board into 4 equal parts:



We can now apply our recursive procedure to the top left board which has a missing square. To be able to apply recursion to the remaining 3 squares we place a domino at the centre as shown below.



We treat the covered squares in each of the three pieces as a missing square and can proceed by recursion applied on all 4 squares whose sides are half the size of the size of the original square.

■

Exercise 2.2. [K] Given positive integers M and n, compute M^n using only $O(\log n)$ many multiplications.

Solution. Note that

$$M^n = \begin{cases} \left(M^{\frac{n}{2}}\right)^2 & \text{if } n \text{ is even} \\ \left(M^{\frac{n-1}{2}}\right)^2 \times M & \text{if } n \text{ is odd.} \end{cases}$$

Hence, we can proceed by divide and conquer. If n is even, we recursively compute $M^{\frac{n}{2}}$ and then square it. If n is odd, we recursively compute $M^{\frac{n-1}{2}}$, square it and then multiply by another M. Since n is (approximately) halved in each recursive call, there are at most $O(\log n)$ recursive calls, and since we perform only one or two multiplications in each recursive call, the algorithm performs $O(\log n)$ many multiplications, as required.

Alternative Solution: Any positive integer n can be uniquely written in base 2, and therefore can be uniquely written as the sum of distinct powers of 2. Thus, M^n can be written the product of powers of M where the index is itself a power of 2, i.e. $M, M^2, M^4, M^8, \ldots$. We can obtain these powers of M in $O(\log n)$ time by repeated squaring, and then multiply together the appropriate powers to get M^n. The appropriate powers to multiply are the powers M^{2^i} such that the ith least significant bit of the binary representation of n is 1. For example, to obtain M^{11}, the binary representation of 11 is $(1011)_2$, and hence we should multiply together M, M^2, and M^8. ■

Exercise 2.3. [H] Suppose you are given an array A containing $2n$ numbers. The only operation that you can perform is make a query if element $A[i]$ is equal to element $A[j]$, $1 \leq i, j \leq 2n$. Your task is to determine if there is a number which appears in A at least n times using an algorithm which runs in linear time.

> **Hint —** *A rather a tricky one !! The reasoning resembles a little bit the reasoning used in the celebrity problem (Tutorial Question 1.5): try comparing them in pairs and first find one or at most two possible candidates and then count how many times they appear.*

Solution. Create two arrays B and C, initially both empty. Repeat the following procedure n times:

- Pick any two elements of A and compare them.
  - If the numbers are different, discard both of them.
  - If instead the two numbers are equal, append one of them to B and the other to C.

> **Claim —** If a value x appears in at least half the entries of A, then the same value also appears in at least half the entries of B.

Proof. Suppose such a value x exists. In a pair of distinct numbers, at most one of them can be x, so after discarding both, x still makes up at least half the remaining array elements. Repeating this, x must account for at least half the values appearing in B and C together. But B and C are identical, so at least the entries of B are equal to x.

We can now apply the same algorithm to B, and continue reducing the instance size until we reach an array of size two. The elements of this array are the only candidates for the original property, i.e., the only values that could have appeared n times in the original array.

There is a special case; B and C could be empty after the first pass. In this case, the only candidates for the property are the values which appear in the last pair to be considered. For each of these, perform a linear scan to find their frequency in the original array, and report the answer accordingly.

Finally we analyse the time complexity. Clearly, each step of the procedure repeated n times above takes constant time, so we can reduce the problem from array A (of length $2n$) to array B (of length $\leq n$) in $\Theta(n)$ time. Letting the instance size be *half* the length of the array (to more neatly fit the Master Theorem), we can express the worst case using the recurrence

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n).$$

Now $a\, f(n/b) = f(n/2)$, which is certainly less than say $0.9f(n)$ for large n as f is linear. By case 3 of the Master Theorem, the time complexity is $T(n) = \Theta(f(n)) = \Theta(n)$, as required. ■

Exercise 2.4. [H] You and a friend find yourselves on a TV game show! The game works as follows. There is a **hidden** $N \times N$ table A. Each cell $A[i, j]$ of the table contains a single integer and no two cells contain the same value. At any point in time, you may ask the value of a single cell to be revealed. To win the big prize, you need to find the N cells each containing the **maximum** value in its row. Formally, you need to produce an array $M[1..N]$ so that $A[r, M[r]]$ contains the maximum value in row r of A, i.e., such that $A[r, M[r]]$ is the largest integer among $A[r, 1], A[r, 2], \ldots, A[r, N]$. In addition, to win, you should ask at most $2N\lceil \log N \rceil$ many questions. For example, if the hidden grid looks like this:

	Column 1	Column 2	Column 3	Column 4
Row 1	**10**	5	8	3
Row 2	1	**9**	7	6
Row 3	-3	**4**	-1	0
Row 4	-10	-9	-8	**2**

then the correct output would be $M = [1, 2, 2, 4]$.
Your friend has just had a go, and sadly failed to win the prize because they asked N^2 many questions which is too many. However, they whisper to you a hint: they found out that M is **non-decreasing**. Formally, they tell you that $M[1] \leq M[2] \leq \cdots \leq M[N]$ (this is the case in the example above).
Design an algorithm which asks at most $\mathbf{O(N \log N)}$ many questions that produces the array M correctly, even in the very worst case.

> **Hint —** *Note that you do not have enough time to find out the value of every cell in the grid! Try determining $M[N/2]$ first, and see if divide-and-conquer is of any assistance.*

Solution. We first find $M[N/2]$. We can do this in N questions by simply examining each element in row $N/2$, and finding the largest one.

Suppose $M[N/2] = x$. Then, we know that $M[i] \leq x$ for all $i < N/2$ and that $M[j] \geq x$ for all $j > N/2$. Thus, we can recurse to solve the same problem on the sub-grids

$$A\left[1..\left(\frac{N}{2} - 1\right)\right][1..x]$$

and

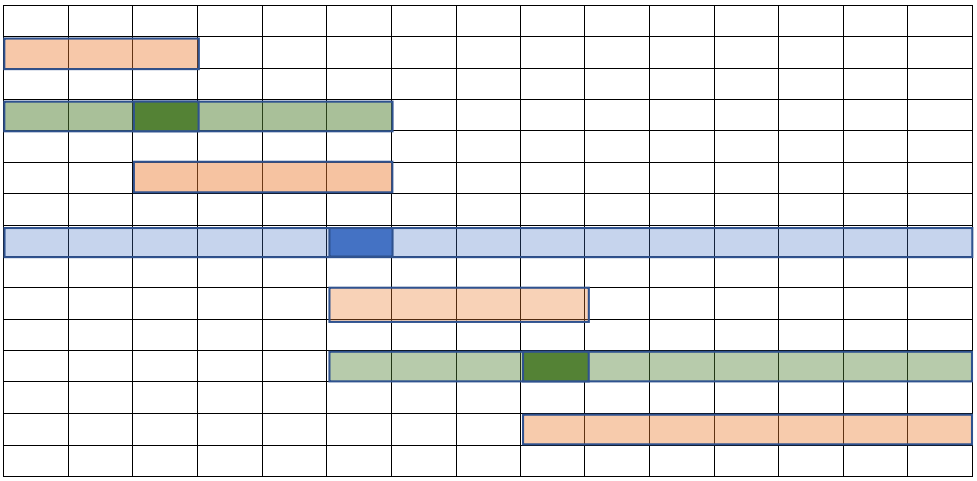
$$A\left[\left(\frac{N}{2} + 1\right)..N\right][x..N].$$

It remains to show that this approach uses at most $2N\lceil \log N \rceil$ questions.

> **Claim —** At each stage of recursion, at most two cells of any column are investigated.

Proof. We observe the following.

- In the first call of the recursion, only one cell of each column has been investigated. These cells are marked in blue, with the maximum in dark blue.
- In the second call of the recursion, we investigate two cells of one column, and one cell of each of the other columns. These cells are marked in green, with the maxima in dark green.



- In the third call of the recursion, we investigate two cells in each of three columns, and one cell of each of the other columns. These cells are marked in orange.

Since the investigated ranges overlap only at endpoints, the claim holds true. ■

So in each recursion call, at most $2N$ cells were investigated. The search space decreases by at least half after each call, so there are at most $\lceil \log_2 N \rceil$ many recursion calls. Therefore, the total number of cells investigated is at most $2N\lceil \log_2 N \rceil$.

Additional exercise: using the above reasoning, try to find a sharper bound for the total number of cells investigated. ■

Exercise 2.5. [H] Define the Fibonacci numbers as

$$F_0 = 0, F_1 = 1, \text{ and } F_n = F_{n-1} + F_{n-2} \text{ for all } n \geq 2.$$

Thus, the Fibonacci sequence is as follows:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots.$$

(a) Show, by induction or otherwise, that

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

for all integers $n \geq 1$.

(b) Hence or otherwise, give an algorithm that finds F_n in $O(\log n)$ time.

Hint — *You may wish to refer to Example 1.5 on page 28 of the "Review Material" booklet.*

Solution.

When $n = 1$ we have

$$\begin{aligned} \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 \end{aligned}$$

so our claim is true for $n = 1$.

Let $k \geq 1$ be an integer, and suppose our claim holds for $n = k$ (Inductive Hypothesis). Then

$$\begin{aligned}\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{k+1} &= \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{k}\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}\\ &= \begin{pmatrix}F_{k+1} & F_k\\ F_k & F_{k-1}\end{pmatrix}\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}\end{aligned}$$

by the Inductive Hypothesis. Hence

$$\begin{aligned}\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{k+1} &= \begin{pmatrix}F_{k+1}+F_k & F_{k+1}\\ F_k+F_{k-1} & F_k\end{pmatrix}\\ &= \begin{pmatrix}F_{k+2} & F_{k+1}\\ F_{k+1} & F_k\end{pmatrix}\end{aligned}$$

by the definition of the Fibonacci numbers. Hence, our claim is also true for $n = k + 1$, so by induction it is true for all integers $n \geq 1$.

Let

$$G = \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}.$$

Now we can use either of the methods from the previous problem.

1. We can use divide and conquer, recursively calculating

$$\begin{cases}\left(G^{\frac{n}{2}}\right)^2 & \text{if } n \text{ is even}\\ \left(G^{\frac{n-1}{2}}\right)^2 G & \text{if } n \text{ is odd.}\end{cases}$$

   At each of $O(\log n)$ steps of the recursion, we multiply either two or three 2-by-2 matrices in constant time, so the algorithm runs in $O(\log n)$.

2. We can instead compute $G, G^2, G^4, \ldots$ by repeated squaring, and multiply those terms which correspond to bits appearing in the binary representation of n. There are $\lfloor \log_2 n \rfloor$ such terms to consider, so the algorithm again takes $O(\log n)$ time.

■

Exercise 2.6. [H] Let A be an array of n integers, not necessarily distinct or positive. Design a $\Theta(n \log n)$ algorithm that returns the maximum sum found in any contiguous subarray of A.

> **Note** — Note that there is an $O(n)$ algorithm that solves this problem; however, the technique used in that solution is much more involved and will be taught in due time.

Solution. Note that brute force takes $\Theta(n^2)$.

The key insight to this problem is the following; once we split the original array in half, the maximum subarray must occur in one of the following cases:

- the maximum subarray sum occurs *entirely* in the left half of the original array.
- the maximum subarray sum occurs *entirely* in the right half of the original array.

- the maximum subarray sum overlaps across both halves.

We can compute the first two cases simply by making recursive calls on the left and right halves of the array. The third case can be computed in linear time. We now fill in the details of the algorithm.

Given an array of size N, split the array into smaller arrays of approximate size $N/2$ each. We can compute the maximum subarray sum in the left half using recursive calls on the left half of the array. Similarly, we can compute the maximum subarray sum in the right half using recursive calls on the right half of the array. To compute the maximum sum in the case where the subarray overlaps across both halves, we find the maximum sum of the subarray beginning at the midpoint and end at some point to the left of the midpoint. We then add this to the maximum sum of the subarray beginning at (midpoint + 1) and end at some point to the right of (midpoint + 1). This takes $O(n)$ time. Our algorithm then returns the maximum of these three cases.

One caveat is that, if the maximum sum is negative, then our algorithm should return 0 to indicate that choosing an *empty* subarray is best. To compute the time complexity of the algorithm, note that we're splitting our problem from size N to two $N/2$ problems and doing an $O(n)$ overhead to combine the solutions together. This gives us the recurrence relation

$$T(n) = 2T(n/2) + O(n).$$

This falls under Case 2 of the Master Theorem, so $T(n) = \Theta(n \log n)$ is the total running time of the algorithm. ■

§3 Karatsuba's trick

Exercise 3.1. [K] Let $P(x) = a_0 + a_1 x$ and $Q(x) = b_0 + b_1 x$, and define $R(x) = P(x) \cdot Q(x)$. Find the coefficients of $R(x)$ using only three products of pairs of expressions each involving the coefficients a_i and/or b_j.
Addition and subtraction of the coefficients do not count towards this total of three products, nor do scalar multiples (i.e. products involving a coefficient and a constant).

Solution. We use (essentially) the Karatsuba trick:

$$\begin{aligned} R(x) &= (a_0 + a_1 x)(b_0 + b_1 x) \\ &= a_0 b_0 + (a_0 b_1 + b_0 a_1)x + a_1 b_1 x^2 \\ &= a_0 b_0 + [(a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1]\, x + a_1 b_1 x^2. \end{aligned}$$

Note that the last expression involves only three multiplications: $a_0 b_0$, $a_1 b_1$ and $(a_0 + a_1)(b_0 + b_1)$. ■

Exercise 3.2. [K] Recall that the product of two complex numbers is given by

$$\begin{aligned} (a + \mathrm{i}b)(c + \mathrm{i}d) &= ac + \mathrm{i}ad + \mathrm{i}bc + ß^2 bd \\ &= (ac - bd) + \mathrm{i}(ad + bc) \end{aligned}$$

(a) Multiply two complex numbers $(a + \mathrm{i}b)$ and $(c + \mathrm{i}d)$ (where a, b, c, d are all real numbers) using only 3 real number multiplications.

(b) Find $(a + \mathrm{i}b)^2$ using only two multiplications of real numbers.

(c) Find the product $(a + \mathrm{i}b)^2 (c + \mathrm{i}d)^2$ using only five real number multiplications.

Solution. **(a)** This is again the Karatsuba trick:

$$\begin{aligned}(a+\mathrm{i}b)(c+\mathrm{i}d) &= ac-bd+(bc+ad)\mathrm{i}\\ &= ac-bd+\left[(a+b)(c+d)-ac-bd\right]\mathrm{i},\end{aligned}$$

so we need only 3 multiplications: ac and bd and $(a+b)(c+d)$.

(b) We again expand and refactorise:

$$\begin{aligned}(a+\mathrm{i}b)^2 &= a^2-b^2+2ab\mathrm{i}\\ &= (a+b)(a-b)+(a+a)b\mathrm{i}.\end{aligned}$$

(c) Observe that

$$(a+\mathrm{i}b)^2(c+\mathrm{i}d)^2 = \left[(a+\mathrm{i}b)(c+\mathrm{i}d)\right]^2.$$

Therefore, we can now use (a) to multiply $(a+\mathrm{i}b)(c+\mathrm{i}d)$ with only three multiplications and then use (b) to square the result with two additional multiplications.

■

Exercise 3.3. [H] Let $P(x) = a_0 + a_1x + a_2x^2 + a_3x^3$ and $Q(x) = b_0 + b_1x + b_2x^2 + b_3x^3$, and define $R(x) = P(x)\cdot Q(x)$. You are tasked to find the coefficients of $R(x)$ using only twelve products of pairs of expressions each involving the coefficients a_i and/or b_j.

Challenge — Can you do better?

Solution. We again use the Karatsuba trick:

$$\begin{aligned}R(x) &= (a_0+a_1x+a_2x^2+a_3x^3)(b_0+b_1x+b_2x^2+b_3x^3)\\ &= (a_0+a_1x)(b_0+b_1x)+(a_0+a_1x)(b_2+b_3x)x^2\\ &+ (a_2+a_3x)(b_0+b_1x)x^2+(a_2+a_3x)(b_2+b_3x)x^4\\ &= a_0b_0+\left[(a_0+a_1)(b_0+b_1)-a_0b_0-a_1b_1\right]x+a_1b_1x^2\\ &+ \left\{a_0b_2+\left[(a_0+a_1)(b_2+b_3)-a_0b_2-a_1b_3\right]x+a_1b_3x^2\right\}x^2\\ &+ \left\{a_2b_0+\left[(a_2+a_3)(b_0+b_1)-a_2b_0-a_3b_1\right]x+a_3b_1x^2\right\}x^2\\ &+ \left\{a_2b_2+\left[(a_2+a_3)(b_2+b_3)-a_2b_2-a_3b_3\right]x+a_3b_3x^2\right\}x^4.\end{aligned}$$

Note that each of the last four lines involves only three multiplications. Expanding and regrouping will yield the coefficients of $R(x)$.

It is actually possible to solve the problem using only seven multiplications! ■

§4 Selection

Exercise 4.1. [K] Suppose we modify the median of medians QUICKSELECT algorithm to use blocks of 7 rather than 5. Determine the appropriate worst-case recurrence, and solve for the asymptotic growth rate.

Solution. We first recurse to find the median of the $n/7$ block medians. That median is guaranteed to be greater than four elements in each of $n/14$ blocks, and less than four elements in each of $n/14$ blocks.

Therefore the worst case partition is in a ratio of $2:5$, and we may have to recursively select from $5n/7$ of the original items. Therefore the recurrence is

$$T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{5n}{7}\right) + \Theta(n).$$

Now we can deduce a pattern that arises from the number of items,

- the 2 subproblems at the second level of the recursion have a total of $6n/7$ items,
- the 4 subproblems at the third level of the recursion have a total of $36n/49$ items,
- and so on.

The total size of all subproblems is bounded above by

$$n\left(1 + \frac{6}{7} + \left(\frac{6}{7}\right)^2 + \ldots\right) = \frac{n}{1-\frac{6}{7}} = 7n,$$

so the total time taken (for forming and sorting blocks as well as partitioning) is $\Theta(n)$. ■

Exercise 4.2. [K] Given two *sorted* arrays A and B, each of length n, design a $O(\log n)$ algorithm to find the (lower) median of all $2n$ elements of both arrays. You may assume that n is a power of two and that all $2n$ values are distinct.

Solution. If $A[n/2] < B[n/2]$, then $A[n/2]$ is smaller than the median, and $B[n/2+1]$ is larger than the median. We can therefore discard the first half of A and the second half of B, and the median of the remaining n elements is our answer.

Similarly, if $A[n/2] > B[n/2]$, we instead discard the second half of A and the first half of B.

We have transformed an instance of size n to an instance of size $n/2$ using only one comparison, so the recurrence is

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1).$$

The critical exponent is $c^* = \log_2 1 = 0$, so by case 2 of the Master Theorem, the asymptotic solution is $T(n) = \Theta(\log n)$. Note that this is the same recurrence as binary search. ■

Exercise 4.3. [K] Given an array A of n distinct integers and an integer $k < \frac{n}{2}$, design an $O(n)$ algorithm which finds the median as well as the k values closest to the median from above and below. You may assume that n is odd.

Solution. Apply median of medians quickselect to find the values k ranks below and k ranks above the median, i.e. the $\left(\frac{n+1}{2} - k\right)$th and $\left(\frac{n+1}{2} + k\right)$th smallest values. Then scan through the array, checking whether each element lies between these bounds. ■

Exercise 4.4. [H] Let A be an array of n distinct integers, each with an associated weight w_i. All w_i are non-negative, and $w_1 + \ldots + w_n = 1$. The *weighted (lower) median* is the element $A[k]$ satisfying

$$\sum_{A[i]<A[k]} w_i < \frac{1}{2} \text{ and } \sum_{A[i]>A[k]} w_i \leq \frac{1}{2}.$$

Design a $O(n)$ algorithm to find the weighted (lower) median.

Solution. The required algorithm is a small modification of median of medians quickselect. After partitioning, we compute the sum of weights of elements left of the pivot, and the sum of weights of those right of the pivot. If either is greater than $\frac{1}{2}$, we recurse on that side. If both are at most $\frac{1}{2}$, we stop.

The additional calculation takes linear time, which is the same as the work already performed to form and sort blocks and partition the array. Therefore, this modification does not worsen the time complexity. ■